SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

A DISTRIBUTED ALGORITHM FOR MINIMUM WEIGHT SPANNING TREES. Revision.

5. TYPE OF REPORT & PERIOD COVERED

Technical Report

6. PERFORMING ORG. REPORT NUMBER

LIDS-P-906-A

7. AUTHOR(s)

R. G. Gallager
P. A. Humblet
P. M. Spira

8. CONTRACT OR GRANT NUMBER(s)

ARPA Order No. 3045/5-7-75
ONR/N00h14-75-C-1183

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Massachusetts Institute of Technology
Laboratory for Information & Decision Systems
Cambridge, Massachusetts 02139

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

Program Code No. 5T10
ONR Identifying No. 049-383

11. CONTROLLING OFFICE NAME AND ADDRESS

Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

12. REPORT DATE

Oct[illegible]79

13. NUMBER OF PAGES

25

14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office)

Office of Naval Research
Information Systems Program
Code 437
Arlington, Virginia 22217

15. SECURITY CLASS. (of this report)

UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

410 950

DDC RECEIVED FEB 20 1980 E

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

N00014-75-C-1183, ARPA Order-3045

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

distributed algorithm spanning tree processors

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A distributed algorithm is presented that constructs the minimum weight spanning tree in a connected undirected graph with distinct edge weights. A processor exists at each node of the graph, knowing initially only the weights of the adjacent edges. The processors obey the same algorithm and exchange messages with neighbors until the tree is constructed. The total number of messages required for a graph of N nodes and E edges is at most $5N \log_2 N + 2E$ and a message contains at most one edge weight plus $\log_2 8N$ bits. The algorithm can be initiated spontaneously at any node or at any subset of nodes.

log of N to the base 2

log of 8N to the base 2

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD A080902

DDC FILE COPY

80 2 19 120 410950

Revised
October 1979

LIDS-P-906-A

A DISTRIBUTED ALGORITHM FOR MINIMUM WEIGHT SPANNING TREES†

by

R. G. Gallager*

P. A. Humblet**

P. M. Spira***

ABSTRACT

A distributed algorithm is presented that constructs the minimum weight spanning tree in a connected undirected graph with distinct edge weights. A processor exists at each node of the graph, knowing initially only the weights of the adjacent edges. The processors obey the same algorithm and exchange messages with neighbors until the tree is constructed. The total number of messages required for a graph of N nodes and E edges is at most $5N \log_2 N + 2E$ and a message contains at most one edge weight plus $\log_2 8N$ bits. The algorithm can be initiated spontaneously at any node or at any subset of nodes.

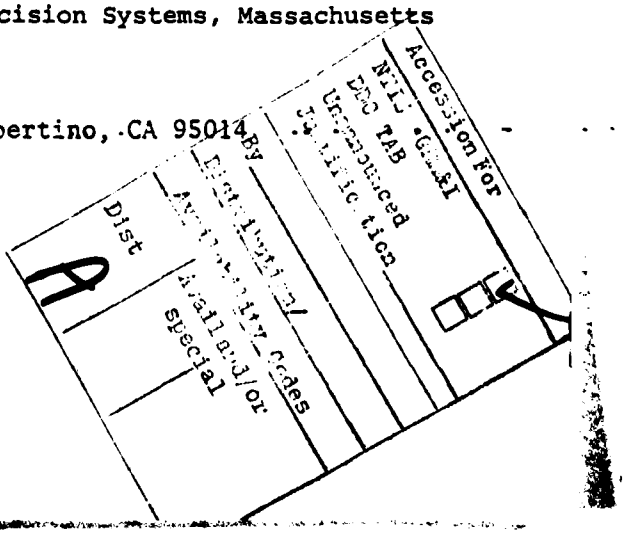
---

†This research was conducted at the M.I.T. Laboratory for Information and Decision Systems with partial support provided by NSF under Grant ENG-77-19971 and by ARPA under GRANT ONR/N00014-75-C-1183.

*Room 35-206, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, MA 02139.

**Room 35-203, same as *.

*** Apple Computer Co., 10260 Bandley Drive, Cupertino, CA 95014



I. Introduction

In this note we consider a connected undirected graph with N nodes and E edges, with a distinct finite weight assigned to each edge. We describe an asynchronous distributed algorithm which determines the minimal weight spanning tree (MST) of the graph. We assume that each node initially knows the weight of each edge adjacent to that node.

Each node performs the same local algorithm, which consists of sending messages over adjoining links, waiting for incoming messages, and processing. Messages can be transmitted independently in both directions on an edge, and arrive after an unpredictable but finite delay, without error and in sequence. After each node completes its local algorithm, it knows which adjoining edges are in the tree and also knows which edge leads to a particular edge designated as the core of the tree.

We view the nodes in the graph as being initially asleep. One or more of the nodes then wake up in any order and start their own local algorithms. The still sleeping nodes will wake up upon receiving messages from awakened neighbors, and will then proceed with their local algorithms.

Under these assumptions, we shall see that the total number of messages exchanged by the nodes to find the MST is less than $2E + 5N \log_2 N$. Each message consists of at most one edge weight, one integer between 0 and $\log_2 N$, and three additional bits.

Our algorithm is similar to an earlier algorithm described by Spira [1], which followed an earlier algorithm given by Dalal [2]. Spira did not analyze in detail the number of messages required by his algorithm, but gave a heuristic argument that the expected number of messages (over randomly

selected graphs) would grow with E and N as E + N log N.

If the nodes of the network have distinct identities that can be ordered, then it is easy to extend the algorithm to the case where the edge weights are not distinct. One simply appends to the edge weight the identities of the two nodes joined by the edge, listing, say the lower ordered node first. These appended weights have the same ordering as before with ties broken by the node identities. If the nodes do not know the identities of their neighbors, then each node can send its identity over each adjoining edge thus requiring a total of 2E extra messages. We have also developed another algorithm, to be briefly described later, that doesn't require distinct weights, and thus doesn't require these additional 2E messages.

If the network has neither distinct edge weights nor distinct node identities, then no distributed algorithm (of the type described above) exists for finding an MST with a bounded number of messages. This can be seen most easily for a three node fully connected graph with equal weight edges. Any two edges form an MST, but since the nodes obey the same algorithms, they have no way to choose. If nodes choose random identities, then the algorithm could be made to work as soon as the identities were all different, but there is no way to guarantee this in a finite number of choices. Naturally the expected number of required choices is small, but any bounded number of messages will fail with some positive probability.

Distributed MST algorithms are useful in communication networks when one wishes to broadcast information from one node to all other nodes and there is a cost associated with each channel of the network. If the cost of using a channel in one direction is different from that in the opposite direction,

then the MST does not provide the desired solution, but a companion paper [5] treats this more general problem. In addition to the broadcast application there are many potential control problems for networks whose communication complexities are reduced by having a known spanning tree. With topology changes caused by possible failures in the network, it is desirable to be able to generate a spanning tree starting from any node or subset of nodes, and the algorithm here is as efficient as any we have been able to find for generating an arbitrary spanning tree. Finally there are a number of applications for distributed algorithms that can select the node in the network with the highest identity number. An efficient distributed algorithm for this problem starts with the MST algorithm and then uses the resulting tree to find the highest numbered node.

II. Review of Spanning Trees

We assume the reader is familiar with the elementary definitions and properties of graphs, paths, cycles, trees, etc., which can be found, for example, in [3], [4]. Suppose that each edge e of a graph has a weight w(e) associated with it. The weight of a tree in the graph is defined as the sum of the weights of the edges in the tree, and our objective is to find a spanning tree of minimum weight, i.e., an MST. A fragment of an MST is defined as a subtree of the MST, i.e., a connected set of nodes and edges of the MST. The algorithm starts with each individual node as a fragment and ends with the MST as a fragment. Define an edge as an outgoing edge of a fragment if one adjacent node is in the fragment and the other is not.

Property 1: Given a fragment of an MST, let e be a minimum weight outgoing edge of the fragment. Then joining e and its adjacent non-fragment node to the fragment yields another fragment of an MST.

Proof: Suppose the added edge e is not in the MST containing the original fragment. Then there is a cycle formed by e and some subset of the MST edges. At least one edge $x \neq e$ of this cycle is also an outgoing edge of the fragment, so that $w(x) \geq w(e)$. Thus deleting x from the MST and adding e forms a new spanning tree which must be minimal if the original tree were minimal. The original fragment with e added is a fragment of the new MST.

Property 2: If all the edges of a connected graph have different weights, then the MST is unique.

Proof: Suppose, to the contrary, that there are two different MST's. Let e be the minimum weight edge that is in one but not both of the trees, and let T be the set of edges of the MST containing e and T' be the edge set of the

other MST. The edge set $\{e\} \cup T'$ must contain a cycle and at least one edge of this cycle, say e', is not in T (since T contains no cycles). Since the edge weights are all different and e' is in one but not both of the trees, $w(e) < w(e')$. Thus $\{e\} \cup T' - \{e'\}$ is the edge set of a spanning tree of smaller weight than T', yielding a contradiction. These properties immediately suggest a general type of algorithm for finding the MST for a graph with different edge weights. One starts with one or more fragments consisting of single nodes. Using property one, these fragments can be enlarged in any order. Whenever two fragments have a common node, property two assures us that the union of these fragments is also a fragment, allowing fragments to be combined into larger fragments. The standard algorithms for generating MST's correspond to different orders in which the above fragments are enlarged and combined. For example, the Prim-Dijkstra algorithm [6], [7] starts with a single node and successively enlarges the fragment until it spans the graph. The Kruskal algorithm [8] starts with all nodes as fragments and sucessively extends the fragment with the smallest weight outgoing edge, combining fragments where possible. Other algorithms [9], [1], [2] start with all nodes as fragments, extend each fragment, then combine, then extend each of the new enlarged fragments, then combine again, and so forth.

The Prim-Dijkstra and Kruskal algorithms work equally well if some of the edge weights are the same. To see this, simply impose an arbitrary ordering on the equal weight edges consistent with the choices made in the execution of the algorithms. Algorithms such as [9], [1], and [2] that extend several fragments without intermediate combining do not necessarily work correctly with equal edge weights. For example, in a three node fully connected network

with equal weight edges, each node could extend with a different edge, giving rise to a cycle when the fragments are combined.

The algorithm to follow is similar to [9], [1], and [2], but is somewhat less synchronized in the sense that each fragment proceeds at its own speed to find its minimum weight outgoing edge and then to combine with the fragment at the other end of that edge. More specifically, we call the fragments consisting of individual nodes level 0 fragments, and for each $i \geq 1$, fragments formed by combining two or more level i-1 fragments are called level i fragments. If the minimum weight outgoing edge from a level i fragment, F, goes to a node in a level i' fragment, F', then if $i < i'$, the fragment F simply joins the fragment F'. If $i = i'$, and if the minimum weight outgoing edge from F' is the same as that from F, the two fragments join into a level i + 1 fragment. Otherwise fragment F simple waits until F' joins some fragment that F can join by the previous rules. This waiting cannot lead to a deadlock, since the fragment of lowest level in the graph, with the smallest minimum weight outgoing edge at that level never has to wait; either the minimum weight outgoing edge goes to a fragment of higher level or to a fragment of the same level with the same minimum weight outgoing edge.

III. Description of the Distributed Algorithm

The algorithm to follow has two major parts: first the nodes in a fragment must co-operatively find the minimum weight outgoing edge from the fragment, and second the fragment must combine into a higher level fragment at the appropriate time. We first describe the structure of a fragment.

Each fragment (other than a single node, level 0 fragment) has an identity known to all the nodes in the fragment. This identity is in fact the weight of a particular edge in the fragment called the fragment core. Each edge of the fragment other than the core has a direction associated with it, the direction of the path in the fragment to the core (see Figure 1). Each node in the fragment has one adjacent

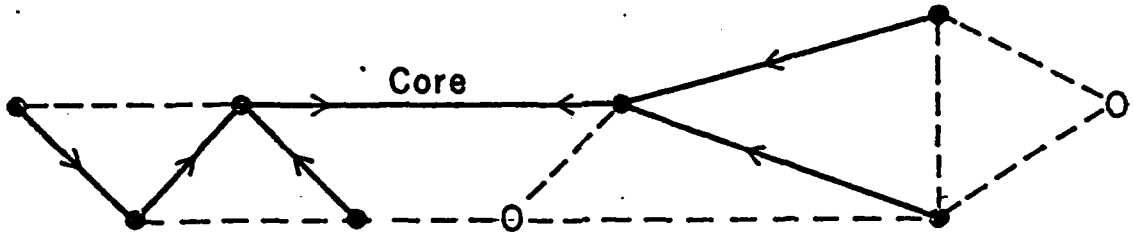


FIGURE 1

FRAGMENT

Fragment edges and nodes are indicated by solid lines and dots; non fragment edges and nodes by dotted lines and circles.

fragment branch directed toward the core, and the node calls this branch the in-branch. The core itself is the in-branch for each of the two adjacent nodes. This structure allows the nodes adjacent to the core to broadcast messages efficiently to the other nodes of the fragment and to collect information from them. We shall first describe the algorithm for fragments at non-zero levels. Zero level fragments will be described later.

A new fragment, at some level i, is formed when two level i-1 fragments combine, both having the same minimum weight outgoing edge. The core of this new fragment then becomes the above minimum weight outgoing edge. The two nodes adjacent to the core then broadcast an Initiate message to the other nodes of the fragment. This message is sent outward on the tree branches and is relayed outward by the intermediate nodes on the tree. The initiate message carries the new fragment level and identity as arguments, providing all nodes in the fragment with this information. The initiate message also contains the node state, which we discuss later. If other fragments at level i-1 are trying to connect in to the nodes of the new level i fragment, the initiate message is passed on to them also, putting them into the new fragment. The initiate message is also passed on to fragments trying to connect into these new nodes, and so forth.

When a node receives this initiate message, it starts to find its minimum weight outgoing edge. The difficulty here is that a node does not know which edges are outgoing. This difficulty is resolved as follows: Each node classifies each of its adjacent edges into one of three possible states: Branch, if the edge is a branch in the current fragment; Rejected, if the edge is not a branch but has been discovered to join two nodes of the fragment; and Basic if the edge is neither a branch nor rejected.

In order to find its minimum weight outgoing edge, a node picks the minimum weight Basic edge and sends a Test message on it. The test message carries the fragment identity and level as arguments. Subject to a few exceptions, the node receiving a test message sends back the message Accept if its own fragment identity differs from that in the message. If the fragment

identities are the same, the node puts the edge in the Rejected state and sends back the message Reject. When a node receives a reject message, it puts the edge in the Rejected state and tests the next best basic edge, continuing until it finds the minimum weight outgoing edge, indicated by an accept message, or until all basic edges are exhausted.

One exception to the above rule for sending accept and reject messages is taken to improve the worst case communication complexity. It a node has both sent and received a test message on an edge with equal fragment identities, then no reject messages are sent and each node rejects the edge and continues testing its next best edge.

Another exception to the above rule arises because of the asynchronous nature of the algorithm; it is necessary to ensure that after a node sends an accept message it does not join the same fragment as the node receiving the accept message. A property of the algorithm, which can be verified by induction on any allowable time ordering of events in the algorithm, is that whenever a node joins a new fragment, the level increases. Thus if the node sending an accept message is at the same or higher level as the recipient, the sender can never become part of the current fragment of the recipient. A node doesn't send an accept message if it is at a lower level than the potential recipient; instead it will put the test message back on the end of the incoming message queue. The message will keep getting returned to the queue until the node is at a higher level. This forces the higher fragment to wait, but cannot cause a deadlock because lower level fragments never have to wait for higher level fragments.

We have just described how each node in a fragment eventually finds its minimum weight outgoing edge, if any. The nodes must now co-operate, by sending Report messages, to find the minimum weight outgoing edge from the entire

fragment; if no node has outgoing edges, the algorithm is complete and the fragment is the MST. In particular, each leaf node of the fragment, i.e., each node adjacent to only one fragment branch, sends the message Report (W) on its inbound branch; W is the weight of the minimum weight outgoing edge from the node, and W is infinity if there are no outgoing edges. Similarly each interior node of the fragment waits until it has both found its own minimum weight outgoing edge and received report messages on all outbound fragment branches. The node then denotes the edge (either outgoing edge or outbound fragment branch) on which the smallest of these weights, W, was found as best-edge, and the node sends Report W on its inbound branch. Eventually the two nodes adjacent to the core send report messages on the core branch itself, allowing each of these nodes to determine both the weight of the minimum weight outgoing edge and the side of the core on which this edge lies.

After the two core nodes have exchanged report messages, the best edges saved by the fragment nodes make it possible to trace the path from the core to the node having the minimum weight outgoing edge. The message Change-core is then sent over each branch of this path, and the inbound edge for each of these nodes is changed to correspond to Best-edge. When this message reaches the node with the minimum weight outgoing edge, the inbound edges form a rooted tree, rooted at this node. Finally this node sends the message Connect (L) over the minimum weight outgoing edge; L is the level of the fragment.

As explained before, at least two fragments at the lowest level, L, must have the same minimum weight outgoing edge, so that the message Connect (L)

will travel in both directions over the edge. At this point, this edge becomes the core of a new fragment at level L+1, and initiate messages are sent out from this core as described earlier. This rule for forming new fragments ensures that a level L+1 fragment always contains at least two level L fragments ($L \geq 0$); it follows that level L fragments contain at least 2^L nodes, and thus that fragment levels are at most $\log_2 N$.

As each node in the new level L+1 fragment receives the level L+1 initiate message, it checks whether it has received connect messages from nodes at level L or less and if so, it responds with an initiate message. Similarly it checks for test messages that can now be responded to. In the algorithm this delayed response to messages is handled by placing messages whose response must be delayed back on the queue; they are then automatically served in the normal way after the receipt of an initiate message.

Level 0 fragments are quite different from other fragments in that they have no core and thus no identity. Finding the minimum weight outgoing edge is trivial, however; it is simply the minimum weight adjacent edge. Thus level 0 fragments can start directly by sending the connect message over the minimum weight edge. We also assume that level 0 fragments can be in a quiescent state called Sleeping. The algorithm is started by one or more nodes spontaneously awaking and sending a connect message as described above. Other nodes wake up on receipt of the first message from another node, which must be either a connect message or a test message. In general, a node can be in one of three possible states--the initial state Sleeping, the state Find while participating in finding the minimum weight outgoing edge, and the state Found at other times.

Finally, consider what happens when a connect message from a node in a low level fragment F reaches a node n' in a higher level fragment F'.

Due to our strategy of never making a low level fragment wait, node n' immediately sends an initiate message to n. If node n' has not yet sent its report message at the given level, fragment F simply joins fragment F' and participates in finding the minimum weight outgoing edge from the enlarged fragment. If, on the other hand, node n' has already sent its report message, then we can deduce that an outgoing edge from node n' has a lower weight than the minimum weight outgoing edge from F. This eliminates the necessity for F to join the search for the minimum weight outgoing edge. These two cases are distinguished by sending the node state, either Find or Found in the initiate message. The nodes in fragment F go into state Find or Found depending on this parameter of the initiate message, and send Test messages only in the Find state.

We now briefly describe a modification of the algorithm that can be used for non-distinct edge weights and that does not require 2E extra messages for appending the adjacent node identities to the edge weight. In the modification, fragments are identified by node identities, which are ordered and distinct. A minimum weight outgoing edge from a fragment is found as before, and a connect message is sent over that edge as before. The new feature is that a connect message on edge e from fragment F to F' is later cancelled if 1) both fragments are at the same level and F > F'; 2) some fragment F" at the same level has sent a connect message to F and F" < F; 3) an initiate message has not already been sent back on edge e. When a connect message is cancelled, the node that sent it increases its level and sends out a new initiate message, in this case joining fragments F and F". This modification also works for multigraphs (graphs with multiple edges joining a pair of nodes), whereas special provisions are required for mulitgraphs otherwise.

IV. Communication Cost Analysis

We determine here an upperbound on the number of messages exchanged during the execution of the algorithm. Note that the most complex message contains one edge weight, one level between 0 and log N, and a few bits to indicate message type.

Since an edge can be rejected only once, and each rejection requires two messages, there are at most 2E test or reject messages leading to rejections.

Next, while a node is at a given level except the zeroth and the last, it can receive at most one initiate and one accept message. It can transmit at most one successful test message, one report message and one change-root or connect message. Since $\log_2 N$ is an upper bound on the highest level, a node can go through at most (-1 + log N) levels not counting the zeroth and last, and this accounts for at most 5N(-1 + log N) messages.

At level 0, each node can receive at most one initiate message and can transmit at most one connect message. At the last level, each node can send at most one report message. This adds less than 5N messages to our grand total which becomes 5N log N + 2E.

Note that if the number of nodes in the graph is initially unknown, as we have implicitly assumed, then no distributed algorithm can find the MST with fewer than E messages; if there is an edge over which no message is sent, then there might have been a node at the center of that edge, causing the algorithm to fail.

V. Timing Analysis

Although it appears that the algorithm typically allows a large amount of parallelism in messages, it is not difficult to find examples such as Figure 2 in which most of the messages are sent sequentially. In particular, if node S originally wakes up, each node j sequentially both sends a test message to each $i \leq j-2$ and receives a reject before any $j' > j$ is awakened. This leads to $N(N-1)$ sequential messages.

If time is important, it is preferable to awaken all nodes originally; this can be done in at most N-1 time units assuming each message transmission requires at most one time unit. We now show that with this assumption of initial awakening, the algorithm requires at most $5N \log_2 N$ time units. Note first that by time N, each node will be awakened and will have sent a connect message. By time 2N, each node must be at level 1 through the propagation of initiate signals.

We proceed by induction on the level numbers showing that it takes at most $5\ell N-3N$ time units until all nodes are at level ℓ. This is true for $\ell = 1$; assume it true for ℓ. At level ℓ, each node can send at most N test messages which will be answered before time $5\ell N - N$. The propagation of the report, change-root and connect, and initiate messages can take at most 3N units, so that by time $5(\ell+1)N - 3N$ all nodes are at level $\ell+1$. At the highest level, $\ell \leq \log_2 N$, only test, reject, and report messages are used, so the algorithm is complete by time $5N \log_2 N$.

A worst case $O(N \cdot \log N)$ is possible, as is shown by the following example (Figure 3), where the handle and the head both contain $\frac{N}{2}$ nodes. If the edge weights in the handle increase as one gets away from the head,

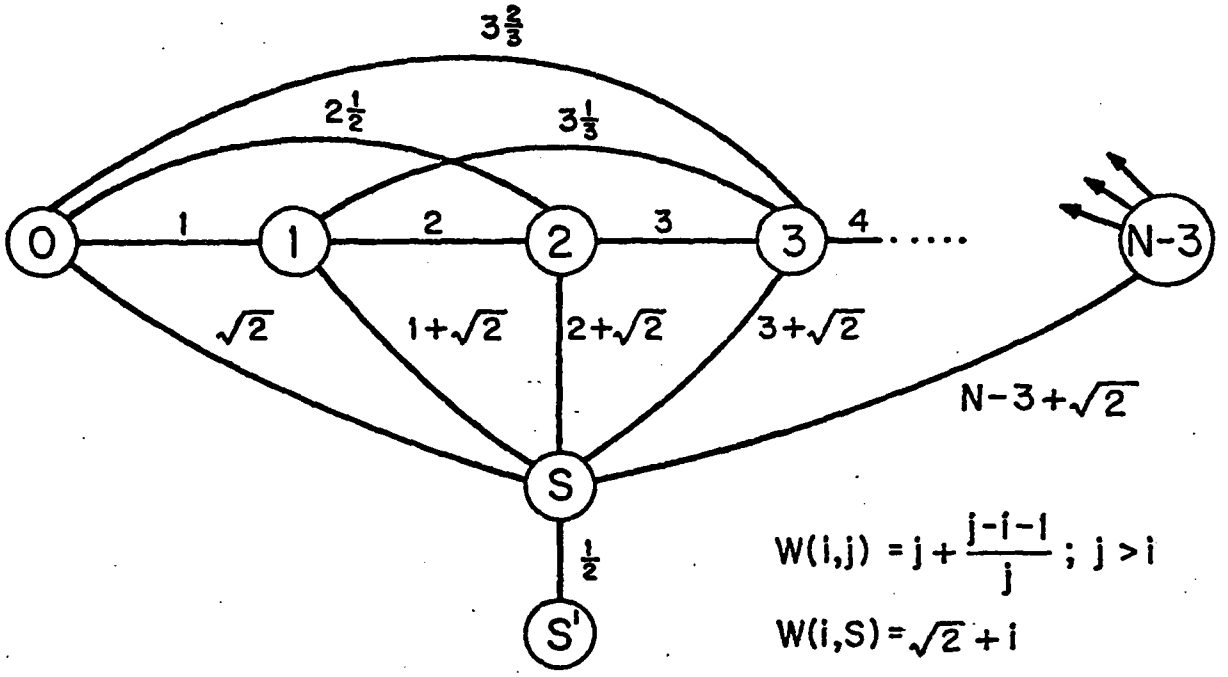


FIGURE 2

Example When Algorithm Requires N(N-1) Message Time Units

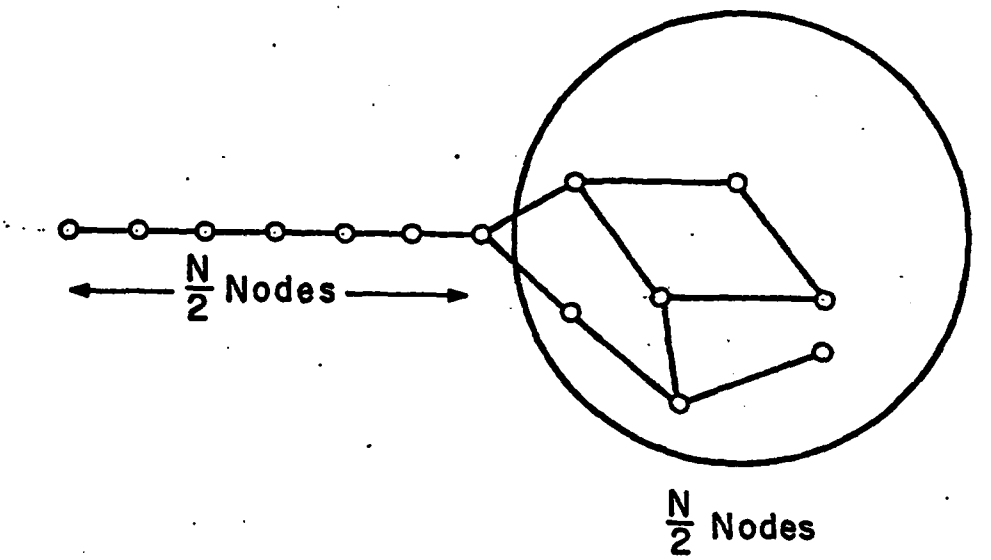


FIGURE 3

Example when $O(N \log_2 N)$ Message Time Units are Required With Initial General Awakening.

then all nodes in the handle will be in the same fragment at level 1. If processing the head requires $\log \frac{N}{2}$ levels and if at each level a fragment joins the handle, which can happen, then the time until completion will be $O(N \log N)$.

References

[1] Spira, P., "Communication Complexity of Distributed Minimum Spanning Tree Algorithms", Proceedings 2nd Berkeley Conf. on Distributed Data Management and Computer Networks, June 1977.

[2] Dalal, Y., "Broadcast Protocols in Packet Switched Computer Networks", Stanford University, Dept. of Electrical Engineering, Technical Report No. 128, April, 1977 (revised version for publication in preparation).

[3] Liu, C.L., Introduction to Combinational Mathematics, McGraw Hill, 1968.

[4] Lawler, E., Combinatorial Optimization-Networks and Matroids, Holt, Rinehart, and Winston, New York, 1976.

[5] Humblet, P.A., "A Distributed Algorithm for Arborescences", in preparation.

[6] Prim, R.C., "Shortest Connection Networks and Some Generalizations", Bell System Tech. Journal, 36, pp. 1389-1401, 1957.

[7] Dijkstra, E., "Two Problems in Connection with Graphs", Num. Math. 1, pp. 269-271, 1959.

[8] Kruskal, J.B., "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem", Proc. Amer. Math. Soc., 7, pp. 48-50, 1956.

[9] Yao, A.C.C., "An O(E log log V) Algorithm for Finding Minimum Spanning Trees", Inf. Proc. Let., 4, pp. 21-23, 1975.

Appendix

The algorithm to follow is obeyed by each node and consists of listing the responses to each type of message that can be generated. In addition the response to a spontaneous awakening of the node is given. Each node is assumed to queue the incoming messages and to respond to them in first come first served order. One particular response is to place the message back on the end of the queue for delayed servicing, but aside from this, each response is completed before the next is started. Each node, of course, maintains its own set of variables, consisting of its state (denoted by SN and assuming possible values Sleeping, Find, and Found), and the state of the adjacent edges. The state of edge j is denoted by SE(j) and can assume the possible values Basic, Branch, Rejected. It is possible for the edge states at the two nodes adjacent to the edge to be temporarily inconsistent. Initially for each node, SN = Sleeping and SE(j) = Basic for each adjacent edge j. Each node also maintains a fragment identity, FN, a level, LN, and variables best-edge, best-wt, test-edge, and in-branch all of whose initial values are immaterial. There is also an initially empty list of edges called find-list and an intially empty first come first serve queue for incoming messages. Finally the weight of each adjacent edge j is denoted w(j).

The Algorithm
(as executed at each node)

1) Response to spontaneous awakening (can occur only at a node in the sleeping state)

```
Execute procedure wakeup
```

2) Procedure wakeup

```
begin let m be adjacent edge of minimum weight;
      SE(m) ← Branch ;
      SN ← Found ;
      send Connect (0) on edge m    end
```

3) Response to receipt of Connect (L) on edge j

```
begin if SN = Sleeping then execute procedure wakeup ;
      if L < LN
        then begin SE(j) ← Branch ;
             send Initiate (LN, FN, SN) on edge j ;
             if SN = Find then
                  find-list ← find-list ∪ {j}    end
        else if SE(j) = Basic
             then place received message on end of queue
             else send Initiate (LN+1, w(j), Find) on edge j   end
```

4) Response to receipt of Initiate (L,F,S) on edge j

```
begin LN ← L; FN ← F; SN ← S; in-branch ← j ;
      best-edge ← nil  best-wt ← ∞ ;

      if S = Find then execute procedure test ;
      for all i ≠ j such that SE(i) = Branch
          do begin send Initiate (L,F,S) on edge i
             if S = Find then find-list ← find-list ∪ {i} end  end
```

5) Procedure test

```
if there are adjacent edges in the state Basic
   then begin test-edge ← the minimum weight adjacent edge in state Basic;
      send Test (LN,FN) on test-edge  end
   else begin test-edge ← nil; execute procedure report  end
```

6) Response to receipt of Test (L,F) on edge j

```
begin if SN = Sleeping then execute procedure wakeup;
      if L > LN then place received message on end of queue
         else if F ≠ FN then send Accept on edge j
              else begin if SE(j) = Basic then SE(j) ← Rejected;
                   if test-edge ≠ j then send Reject on edge j
                      else execute procedure test end  end
```

7) Response to receipt of Accept on edge j

```
begin test-edge ← nil ;
      if w(j) < best-wt
         then begin best-edge ← j; best-wt ← w(j) end;
      execute procedure report end
```

8) Response to receipt of Reject on edge j

```
begin if SE(j) = Basic then SE(j) ← Rejected;
      execute procedure test end
```

9) Procedure report

```
if find-list = nil and test-edge = nil
   then begin SN ← Found;
        send Report (best-wt) on in-branch end
```

10) Response to receipt of Report(w) on edge j

```
if j ≠ in-branch
   then begin find-list ← find-list - {j};
        if w < best-wt then begin best-wt ← w; best-edge ← j end;
        execute procedure report end
   else if SN = Find then place received message on end of queue
        else if w > best-wt
                then execute procedure change-root
                else if w = best-wt = ∞ then halt
```